# Cassandra

teddyma

Published
with GitBook

# Table of Contents

# 学习Cassandra

本书循序渐进的引导开发人员理解Cassandra是什么，如何工作以及如何使用Apache Cassandra 2.0的功能。

本书更多的关注开发人员的视角，也就是说，不会介绍太多关于Cassandra安装和管理的内容，这些内容可以独立作为一个的针对DBA的话题进行讨论。

## 开源协议

# 关于Cassandra

这一章我们将大致介绍Cassandra能做什么。

Apache Cassandra是一个超高扩展性的开源NoSQL数据库。

Cassandra是当同时需要高扩展性、高可用性、高性能、跨多数据中心和在云端管理超大量结构化、半结构化和非架构化数据时的理想解决方案。

Cassandra具有持续可用性， 线性扩展性，易于管理大量服务器和不会有单点故障等特性。

Cassandra的数据模型支持非常方便的列索引，高性能的类日志结构（log-structured）数据更新, 强大的非规格化（denormalization）和物化（materialized）视图和缓存功能。

# Cassandra的历史

[1]Apache Cassandra最早是Facebook为了改进他们的Inbox搜索功能，由Avinash Lakshman（Amazon Dynamo的作者之一）和Prashant Malik写的。2008年七月成为Google Code上的开源项目。2009年三月成为Apache Incubator项目。2010年2月17日升级为Apache的顶级项目。

成为顶级项目之后的版本更新记录：

- 0.6, released Apr 12 2010, added support for integrated caching, and Apache Hadoop MapReduce
- 0.7, released Jan 08 2011, added secondary indexes and online schema changes
- 0.8, released Jun 2 2011, added the Cassandra Query Language (CQL), self-tuning memtables, and support for zero-downtime upgrades
- 1.0, released Oct 17 2011, added integrated compression, leveled compaction, and improved read performance
- 1.1, released Apr 23 2012, added self-tuning caches, row-level isolation, and support for mixed ssd/spinning disk deployments
- 1.2, released Jan 2 2013, added clustering across virtual nodes, inter-node communication, atomic batches, and request tracing
- 2.0, released Sep 4 2013, added lightweight transactions (based on the Paxos consensus protocol), triggers, improved compactions, CQL paging support, prepared statement support, SELECT column alias support

## 参考


1. http://en.wikipedia.org/wiki/Apache_Cassandra

# CAP定理

[1]CAP定理, 又称布鲁斯定理, 它指出一个分布式计算机系统无法同时满足下面三点：

- 一致性（所有节点在同一时间返回相同的数据）
- 可用性 （保证对每个客户端请求无论成功与否都有响应）
- 分区容忍性（系统中任意信息的丢失或失败不会影响系统的继续运行）

根据该定理，任何一个分布式系统只能满足以上三点中的两点而不可能满足全部满足三点。

# Cassandra & CAP

Cassandra一般被认为是满足AP的系统，也就是说Cassandra认为可用性和分区容忍性比一致性更重要。不过Cassandra可以通过调节replication factor和consistency level来满足C。

# 参考

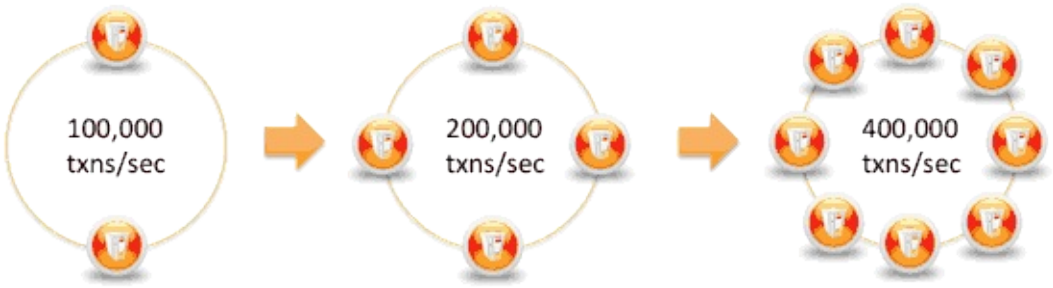

1. http://en.wikipedia.org/wiki/CAP_theorem

# Cassandra能做什么？

Cassandta采用去中心化的架构，也就是说所有的节点都是平等的。 Cassandra自动化地在环或者说数据库集群的所有节点之间进行数据分发。开发人员或者管理员无法，也没有必要通过程序来控制数据分发，因为集群里所有节点的数据分区对用户是透明的。

Cassandra提供了内建的可定制的replication机制，在整个集群的节点上保存冗余的数据副本。也就是说，如果集群上的任意节点发生故障，集群上的其他节点还有一个或者多个副本。Replication可以设置成只在本地的数据中心之间进行，也可以设置成多个数据中心之间或者多个云端区域之间。

[1]Cassandra具有线性扩展性，也就是说可以简单的在线添加新节点从而增加集群的处理能力。例如，如果2个节点可以每秒处理100,000个事务，4个节点就能每秒处理200,000个事务，而8个节点就能每秒处理400,000个事务：




## 参考

1. http://www.datastax.com/documentation/cassandra/2.0/cassandra/gettingStartedCassandraIntro.html

# 数据模型 & CQL

本章将介绍Cassandra的数据模型，CQL，索引以及CQL如何映射到Cassandra的内部数据结构。

# 内部数据结构

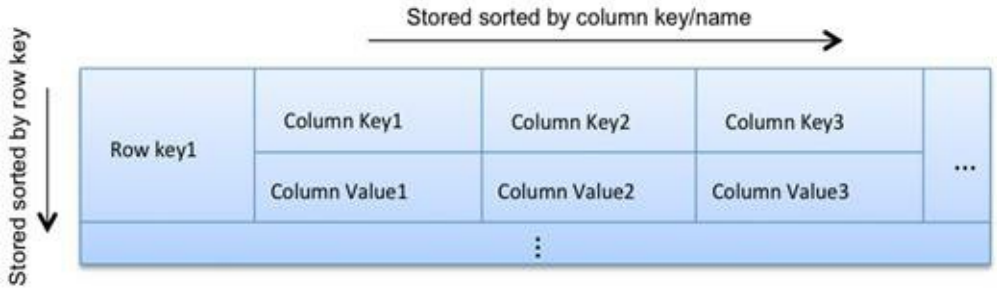
[1]Cassandra的数据模型以列为中心。也就是说，不需要像关系型数据库那样事先定义一个表的所有列，每一行甚至可以包含不同名称的列。

[2]Cassandra的数据模型由keyspaces (类似关系型数据库里的database)，column families（类似关系型数据库里的table), 主键（keys）和列（columns）组成。

| Relational Model | Cassandra Model |
| --- | --- |
| Database | Keyspace |
| Table | Column Family (CF) |
| Primary key | Row key |
| Column name | Column name/key |
| Column value | Column value |

对于每一个column family，不要想象成关系型数据库的表，而要想像成一个多层嵌套的排序散列表（Nested sorted map）。这样能更好地理解和设计Cassandra的数据模型。

```
Map<RowKey, SortedMap<ColumnKey, ColumnValue>>
```




散列表能提供高效的键值查询，而排序的键值能提供高效率的范围查询能力。在Cassandra里，我们可以使用row key和column key做高效的键值查询和范围查询。每一行的列的数量最多允许多达20亿，换句话说，可以拥有所谓的宽行。

列的名称可以直接包含数据，换句话说，有的列可以只有列名没有列值。

# 参考


1. http://www.datastax.com/docs/0.8/ddl/index
2. http://www.bodhtree.com/blog/2013/12/06/my-experience-with-cassandra-concepts/

# CQL

[1]CQL由类似SQL的语句组成，包括修改，查询，保存，变更数据的存储方式等等功能。每一行语句由分号（;）结束。

例如，下面是一个合法的CQL：

```
SELECT * FROM MyTable;

UPDATE MyTable
  SET SomeColumn = 'Some Value'
  WHERE columnName = 'Something Else';
```

这里一共两条语句。每条语句可以写成一行也可以写成多行。

## 大小写

在CQL里，keyspace，column和table的名称是忽略大小写的，除非用双引号（"）括起来，才是大小写敏感的。如果不用双引号括起来，即使CQL写成大写，也会被保存为小写。

例如：

```
CREATE TABLE test (
  Foo int PRIMARY KEY,
  "Bar" int
);
```

CQL的关键字都是忽略大小写的。例如，关键字SELECT和select是等价的。

# CQL数据类型

[2]CQL内建了以下这些列的数据类型:

| CQL类型 | 常量类型 | 说明 |
| --- | --- | --- |
| ascii | strings | US-ASCII字符串 |
| bigint | integers | 64位有符号long |
| blob | blobs | 任意的16进制格式的bytes |
| boolean | booleans | true或false |
| counter | integers | 分布式counter值 (64位long) |
| decimal | integers, floats | 可变精度浮点数 |
| double | integers, floats | 64位IEEE-754浮点数 |
| float | integers, floats | 3位IEEE-754浮点数 |
| inet | strings | IPv4或IPv6格式的IP地址 |
| int | integers | 32位有符号整数 |
| list<T> | n/a | 有序集合, T可以是任意非集合CQL数据类型，例如：int, text等 |
| map<K,V> | n/a | 哈希表, K和V可以是任意非集合CQL数据类型，例如：int, text等 |
| set<T> | n/a | 无序集合, T可以是任意非集合CQL数据类型，例如：int, text等 |
| text | strings | UTF-8编码字符串 |
| timestamp | integers, strings | 日期+时间 |

| | | |
|---|---|---|
| uuid | uuids | 标准UUID格式 |
| timeuuid | uuids | Type 1 UUID |
| varchar | strings | UTF-8编码字符串 |
| varint | integers | 任意精度整数 |

# CQL命令参考

完整的CQL命令参考，请参照：CQL commands.

# 参考


1. http://www.datastax.com/documentation/cql/3.1/cql/cql_reference/cql_lexicon_c.html
2. http://www.datastax.com/documentation/cql/3.1/cql/cql_reference/cql_data_types_c.html

# CQL & 数据架构

从Cassandra 0.7开始，推荐使用CQL来创建column family结构。 同时Cassandra鼓励开发人员分享column family结构。为什么呢？因为，虽然CQL看起来和SQL很像，他们的内部原理完全不同。记住，不要把一个column family想像成关系型数据库的表，而要想像成一个嵌套的键值有序的哈希表。

## 简单例子

例如，用CQL创建一个ColumnFamily(Table)：

```
CREATE TABLE example (
   field1 int PRIMARY KEY,
   field2 int,
   field3 int);
```

然后插入一些行：

```
INSERT INTO example (field1, field2, field3) VALUES (1,2,3);
INSERT INTO example (field1, field2, field3) VALUES (4,5,6);
INSERT INTO example (field1, field2, field3) VALUES (7,8,9);
```

数据怎么存储呢？

```
RowKey: 1
=> (column=, value=, timestamp=1374546754299000)
=> (column=field2, value=00000002, timestamp=1374546754299000)
=> (column=field3, value=00000003, timestamp=1374546754299000)
-------------------
RowKey: 4
=> (column=, value=, timestamp=1374546757815000)
=> (column=field2, value=00000005, timestamp=1374546757815000)
=> (column=field3, value=00000006, timestamp=1374546757815000)
-------------------
RowKey: 7
=> (column=, value=, timestamp=1374546761055000)
=> (column=field2, value=00000008, timestamp=1374546761055000)
=> (column=field3, value=00000009, timestamp=1374546761055000)
```

对于每一个插入的行，有三点需要注意：row key (RowKey: <?>)，列名(column=<?>)和列值(value=<?>)。从上面的例子，我们可以做出一些基本的关于CQL如何映射到Cassandra内部存储结构的总结：

- row key的值是CQL中定义的主键的值。（在CQL术语里面，row key被称作“partition key”）
- CQL中的非主键字段的字段名，映射到内部的列名，字段值映射到内部的列值
- 每一个*RowKey*下的左右字段，是按列名排序存储的

你可能也注意到了，每一个row，都有一个列，列名和列值都为空。这并不是bug。事实上这是为了支持只有row key字段有值，而没有任何其他列有值的情况。

## 更复杂的例子

一个更复杂一点的例子:

```
CREATE TABLE example (
    partitionKey1 text,
    partitionKey2 text,
    clusterKey1 text,
    clusterKey2 text,
    normalField1 text,
    normalField2 text,
    PRIMARY KEY (
        (partitionKey1, partitionKey2),
        clusterKey1, clusterKey2
        )
    );
```

这里我们用字段在内部存储时的类型来命名字段。而且我们已经包含了所有情形。这里的主键不仅仅是复合主键，而且是复合partition key （在PRIMARY KEY部分的前半段，用括号括起的部分）和复合cluster key。

然后插入一些行:

```
INSERT INTO example (
    partitionKey1,
    partitionKey2,
    clusterKey1,
    clusterKey2,
    normalField1,
    normalField2
    ) VALUES (
    'partitionVal1',
    'partitionVal2',
    'clusterVal1',
    'clusterVal2',
    'normalVal1',
    'normalVal2');
```

数据怎么存储呢？

```
RowKey: partitionVal1:partitionVal2
=> (column=clusterVal1:clusterVal2:, value=, timestamp=1374630892473000)
=> (column=clusterVal1:clusterVal2:normalfield1, value=6e6f726d616c56616c31, timestamp=1374630892473000)
=> (column=clusterVal1:clusterVal2:normalfield2, value=6e6f726d616c56616c32, timestamp=1374630892473000)
```

- 注意*partitionVal1*和*partitionVal2*，我们可以发现*RowKey*（也称为partition key）是这两个字段值的组合
- *clusterVal1*和*clusterVal2*这两个cluster key的值（注意是值不是字段名称）的组合，成为了每一个非主键列名的前缀
- 非主键列的值，比如*normalfield1*和*normalfield2*的值，是列名加上cluster key的值之后的列的值
- 每一个row中的列，是按列名排序的，而因为cluster key的值成为非主键列名的前缀，每个row下的所有的列，实际上首先按照cluster key的值排序，然后再按照CQL里的列名排序

# Map, List & Set

下面是一个set，list和map类型的例子：

```
CREATE TABLE example (
    key1 text PRIMARY KEY,
    map1 map<text,text>,
    list1 list<text>,
    set1 set<text>
    );
```

插入数据：

```
INSERT INTO example (
   key1,
   map1,
   list1,
   set1
   ) VALUES (
   'john',
   {'patricia':'555-4326','doug':'555-1579'},
   ['doug','scott'],
   {'patricia','scott'}
   )
```

数据怎么存储呢？

```
RowKey: john
=> (column=, value=, timestamp=1374683971220000)
=> (column=map1:doug, value='555-1579', timestamp=1374683971220000)
=> (column=map1:patricia, value='555-4326', timestamp=1374683971220000)
=> (column=list1:26017c10f48711e2801fdf9895e5d0f8, value='doug', timestamp=1374683971220000)
=> (column=list1:26017c12f48711e2801fdf9895e5d0f8, value='scott', timestamp=1374683971220000)
=> (column=set1:'patricia', value=, timestamp=1374683971220000)
=> (column=set1:'scott', value=, timestamp=1374683971220000)
```

map，list和set的内部存储各不相同：

- 对于map，每一个map的元素成为一列，列名是map字段的列名和这个元素的键值的组合，列值就是这个元素的值
- 对于list，每一个list的元素成为一列，列名是list字段的列名和一个代表元素在list里的index的UUID的组合，列值就是这个元素的值
- 对于set，每一个set的元素成为一列，列名是set字段的列名和这个元素的值的组合，列值总是空值

# References


1. http://www.opensourceconnections.com/2013/07/24/understanding-how-cql3-maps-to-cassandras-internal-data-structure/
2. http://www.opensourceconnections.com/2013/07/24/understanding-how-cql3-maps-to-cassandras-internal-data-structure-sets-lists-and-maps/

# 数据保存在哪里？

对于每一个column family的数据一共有3个层次的数据存储：memtable，commit log 和SSTable.

[1]为了更高的效率，Cassandra在memtable和SSTable里存储数据时不存储列名。例如，如果用下面的CQL插入数据：

```
INSERT INTO k1.t1 (c1) VALUES (v1);
INSERT INTO k2.t1 (c1, c2) VALUES (v1, v2);
INSERT INTO k1.t1 (c1, c3, c2) VALUES (v4, v3, v2);
```

在memtable里, Cassandra保存了下面的数据:

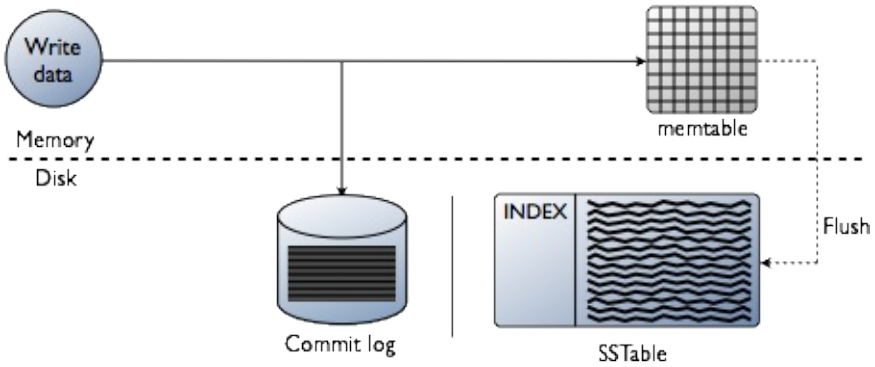
```
k1 c1:v4 c2:v2 c3:v3
k2 c1:v1 c2:v2
```

在磁盘上的commit log里，Cassandra保存了下面的数据:

```
k1, c1:v1
k2, c1:v1 C2:v2
k1, c1:v4 c3:v3 c2:v2
```

在磁盘上的SSTable里，Cassandra在memtable被刷新的时候保存下面的数据：

```
k1 c1:v4 c2:v2 c3:v3
k2 c1:v1 c2:v2
```




# 写路径

当一个写操作发生时，Cassandra将数据保存到内存中的memtable，并且append到磁盘上的commit log。

每个节点上的memtable是当前分区的回写（write-back）缓存，Cassandra从memtable按partition key查询数据。一个column family的数据使用的越多，memtable就越大。Cassandra会为memtable动态分配合适大小的内存，当然你也可以自己管理和最调节memtable的内存分配。和立即写入（write-through）型的缓存不同，memtable会一直保存写入的数据，知道达到配置的阈值，才会被刷新。

当memtable保存的数据超过配置的阈值时，包括索引在内的memtable数据会被加入一个队列，依次被清空和保存到磁盘。刷新数据的时候，Cassandra会先按partition key对memtable中的数据进行排序，然后顺序地（sequentially）保存到磁盘。整个处理过程超级快，因为只有commit log的append和顺序的写入磁盘上的SSTable。

memtable中的数据在被写入SSTable之后，commit log会被清空。commit log的作用是，当发生硬件故障时，用来自动重建memtable中还没有保存到SSTable的数据。

SSTables是不可变的，也就是说，当memtable被刷新和保存为一个SSTable文件之后，SSTable不会被再次写入。因此，一个分区一般会被保存为多个SSTable文件。所以，如果某一行数据不在memtable里，对这行数据的读写需要遍历所有的SSTable。这也是为什么Cassandra的读要比写慢的原因。

# 数据压缩（Compaction）

为了提升读的性能和释放磁盘空间，Cassandra会周期性地通过合并多个SSTable文件中相同partition key的数据的方式进行做数据压缩。

[2]Cassandra内建了两种数据压缩策略： SizeTieredCompactionStrategy和LeveledCompactionStrategy.

- SizeTieredCompactionStrategy适用于写操作更多的情况
- LeveledCompactionStrategy适用于读操作更多的情况

关于数据压缩策略的更多介绍，参见When to Use Leveled Compaction和Leveled Compaction in Apache Cassandra.

# 参考


1. http://www.datastax.com/documentation/cassandra/2.0/cassandra/dml/dml_stores_data_c.html
2. http://www.datastax.com/documentation/cassandra/2.0/cassandra/dml/dml_write_path_c.html

# 索引

[1]Cassandra的索引(index，又称secondary index)支持通过非主键字段查询数据的能力。事实上，如果没有为一个普通字段建索引，Cassandra是完全不支持按这个字段值做任何条件查询的。

一个索引的数据，保存在一个隐藏的column family中。一个节点上的索引，只包含对当前节点存在的本地数据的索引，也就是说，索引数据是不会被replicate到其他节点的。同时，这也意味着，按索引字段进行查询，查询请求需要被转发到所有的节点，并返回合并的查询结果。所以，节点越多，索引查询会越慢。

注意：

直到当前版本（2.0.7）的Apache Cassandra，对于索引字段的条件查询只支持相等比较。不支持对于索引字段的范围查询和字段排序。主要的原因是，保存索引数据的隐藏column family的主键是无序存储的。

# 何时使用索引？

[2]Cassandra的索引功能最适合很多行包含相同的被索引列的值的情形。某一个字段的值，对于所有的行来说，越唯一，索引查询的性能和维护成本就越差。例如：假设有一个播放列表表，包含100万首歌，并且需要按歌手字段查询。因为每个歌手会有很多歌，歌手这个字段，就适合创建索引。

# 何时不要使用索引？

[2]下面这些情形不要使用索引：

- 字段的值对所有的行来说非常唯一，比如时间字段
- 包含counter类型字段的表
- 频繁更新和删除的表
- 在超大分区查询得到很少行的数据（除非先按照其他条件缩小查询范围）

# 参考


1. http://www.datastax.com/documentation/cql/3.1/cql/ddl/ddl_primary_index_c.html
2. http://www.datastax.com/documentation/cql/3.1/cql/ddl/ddl_when_use_index_c.html

# 数据Replication

这一章，我们将介绍数据如何在Cassandra的节点之间进行分发和replicate。

[1]Cassandra是一个点对点系统，所以，创建数据副本和数据的分发都是在一组节点之间互相进行的。

数据是按照column family（table）进行组织。一行数据，通过这一行的partition key唯一标识。 Cassandra使用一致性哈希（consistent hashing）机制在集群里分发数据。每个集群，配置了一个分区器（partitioner）来对每个partition key进行哈希运算。哈希运算的结果决定了这一行数据应该被保存到哪个节点。

一行数据的一个副本被称作一个replica。数据第一次被保存，也被称作创建数据的第一个replica。

# 参考


1. http://www.datastax.com/documentation/cassandra/2.0/cassandra/architecture/architectureDataDistributeAbout_c.html

# 分区器（Partitioners）

[1]分区器决定了数据如何在集群内被分发。简单来说，一个分区器就是一个用来计算partition key哈希值的一个哈希函数。每一行数据，由partition key的值唯一标识，并且根据partition key的哈希值决定如何在集群内被分发。

一个集群，有一个全局唯一的分区器的配置。Cassandra的默认的分区器是Murmur3Partitioner，它一般能满足绝大多数情况的需要。

**Cassandra**提供了以下这些分区器：

- Murmur3Partitioner（默认）：基于MurmurHash哈希算法
- RandomPartitioner：基于MD5哈希算法
- ByteOrderedPartitioner：根据partition key的bytes进行有序分区

Murmur3Partitioner和RandomPartitioner都使用哈希值来平均分配一个column family的数据到集群上的所有节点。

[2]ByteOrderedPartitioner用于基于partition key的bytes进行有序分区。它允许按照partition key进行条件范围查询。也就是说，可以像关系型数据库的主键那样，通过游标，有序遍历所有的partition key。例如，如果使用用户名作为partition key，可以范围查询所有名字在Jake和Jeo之间的用户。这样的条件范围查询，在使用随机分区器时，是不允许的。

尽管范围查询听上去很有吸引力，并不推荐使用有序的分区器，因为：

- 负载均衡更困难
- 顺序的分区容易造成热点（hotspot）
- 多个表的数据的负载均衡不平均

# 参考


1. http://www.datastax.com/documentation/cassandra/2.0/cassandra/architecture/architecturePartitionerAbout_c.html#concept_ds_dwv_npf_fk
2. http://www.datastax.com/documentation/cassandra/2.0/cassandra/architecture/architecturePartitionerBOP_c.html

# Replication策略

[1]Cassandra在多个节点存储replica来确保可靠性和容错性。Replication策略决定了replica保存到哪些节点。

replica的总数由replication factor控制。replication factor 1表示一行数据，只会有一个replica，被保存在唯一一个节点上。replication factor 2表示一行数据有两个副本，每个副本保存在不同的节点上。所有的replica同等重要；没有主次replica之分。一般的规则是，replication factor不应该超过集群里的节点数量。不过，可以先增大replication factor，再添加更多节点。如果replication factor配置超过节点数量，写操作会被拒绝。读操作不受影响。

**Cassandra**内置了两种类型的**replication**策略：

- SimpleStrategy：用于单个数据中心。如果有可能以后会有多个数据中心，应该用NetworkTopologyStrategy
- NetworkTopologyStrategy：对绝大多数部署方式，都强烈推荐该策略，因为今后的扩展更容易

关于每个数据中心应该配置几个**replica**，一般主要考虑以下两个因素：

- 保证读操作没有跨数据中心的延时损耗
- 如何处理硬件故障的情形

对于多数据中心，最常用的两种配置**replica**的策略是：

- 每个数据中心2个replica：基于这种配置，对于每个数据中心，即使单个节点故障，还是能够支持consistency level ONE的本地读
- 每个数据中心3个replica：基于这种配置，对于每个数据中心，即使单个节点故障，还是能够支持consistency level LOCAL_QUORUM的本地读；即使2两个节点故障，还是能够支持consistency level ONE的本地读

# 参考


1. http://www.datastax.com/documentation/cassandra/2.0/cassandra/architecture/architectureDataDistributeReplication_c.html

# 可调节的一致性

[1]一致性指一行数据的所有replica是否最新和同步。Cassandra扩展了最终一致性的概念，对一个读或者写操作，所谓可调节的一致性的概念，指发起请求的客户端，可以通过consistency level参数，指定本次请求，需要的一致性。

# 写操作的Consistency Level

写操作的consistency level指定了写操作在通知客户端请求成功之前，必须确保已经成功完成写操作的replica的数量。

| 级别 | 描述 | 用法 |
| --- | --- | --- |
| ANY | 任意一个节点写操作已经成功。如果所有的replica节点都挂了，写操作还是可以在记录一个hinted handoff事件之后，返回成功。如果所有的replica节点都挂了，写入的数据，在挂掉的replica节点恢复之前，读不到。 | 最小的延时等待，并且确保写请求不会失败。相对于其他级别提供最低的一致性和最高的可用性。 |
| ALL | 写操作必须将指定行的数据写到所有replica节点的commit log和memtable。 | 相对于其他级别提供最高的一致性和最低的可用性。 |
| EACH_QUORUM | 写操作必须将指定行的数据写到每个数据中心的quorum数量的replica节点的commit log和memtable。 | 用于多数据中心集群严格的保证相同级别的一致性。例如，如果你希望，当一个数据中心挂掉了，或者不能满足quorum数量的replica节点写操作成功时，写请求返回失败。 |
| LOCAL_ONE | 任何一个本地数据中心内的replica节点写操作成功。 | 对于多数据中心的情况，往往期望至少一个replica节点写成功，但是，又不希望有任何跨数据中心的通信。LOCAL_ONE正好能满足这样的需求。 |
| LOCAL_QUORUM | 本地数据中心内quorum数量的replica节点写操作成功。避免跨数据中心的通信。 | 不能和SimpleStrategy一起使用。用于保证本地数据中心的数据一致性。 |
| LOCAL_SERIAL | 本地数据中心内quorum数量的replica节点有条件地（conditionally）写成功。 | 用于轻量级事务（lightweight transaction）下实现linearizable consistency，避免发生无条件的（unconditional）更新。。 |
| ONE | 任意一个replica节点写操作已经成功。 | 满足大多数用户的需求。一般离coordinator节点具体最近的replica节点优先执行。 |

注意：

即使指定了consistency level ON或LOCAL_QUORUM，写操作还是会被发送给所有的replica节点，包括其他数据中心的里replica节点。consistency level只是决定了，通知客户端请求成功之前，需要确保写操作成功的replica节点的数量。

# 读操作的Consistency Level

| 级别 | 描述 | 用法 |
| --- | --- | --- |
| ALL | 向所有replica节点查询数据，返回所有的replica返回的数据中，timestamp最新的数据。如果某个replica节点没有响应，读操作会失败。 | 相对于其他级别，提供最高的一致性和最低的可用性。 |
| EACH_QUORUM | 向每个数据中心内quorum数量的replica节点查询数据，返回时间戳最新的数据。 | 同LOCAL_QUORUM。 |
| LOCAL_SERIAL | 同SERIAL，但是只限制为本地数据中心。 | 同SERIAL。 |
| LOCAL_QUORUM | 向每个数据中心内quorum数量的replica节点查询数据，返回时间戳最新的数据。避免跨数据中心的通信。 | 使用SimpleStrategy时会失败。 |
| LOCAL_ONE | 返回本地数据中心内离coordinator节点最近的replica节点的数据。 | 同写操作Consistency level中该级别的用法。 |

| | | |
|---|---|---|
| ONE | 返回由snitch决定的最近的replica返回的结果。默认情况下，后台会触发read repair确保其他replica的数据一致。 | 提供最高级别的可用性，但是返回的结果不一定最新。 |
| QUORUM | 读取所有数据中心中quorum数量的节点的结果，返回合并后timestamp最新的结果。 | 保证很强的一致性，虽然有可能读取失败。 |
| SERIAL | 允许读取当前的（包括uncommitted的）数据，如果读的过程中发现uncommitted的事务，则commit它。 | 轻量级事务。 |
| TWO | 返回两个最近的replica的最新数据。 | 和ONE类似。 |
| THREE | 返回三个最近的replica的最新数据。 | 和TWO类似。 |

# 关于**QUORUM**级别

QUORUM级别确保数据写到指定quorum数量的节点。一个quorum的值由下面的公式四舍五入计算而得：

(sum_of_replication_factors / 2) + 1

sum_of_replication_factors指每个数据中心的所有replication_factor设置的总和。

例如，如果某个单数据中心的replication factor是3，quorum值为2-表示集群可以最多容忍1个节点down。如果replication factor是6，quorum值为4-表示集群可以最多容忍2个节点down。如果是双数据中心，每个数据中心的replication factor是3，quorum值为4-表示集群可以最多容忍2个节点down。如果是5数据中心，每个数据中心的replication factor of 3，quorum值为8 。

如果想确保读写一致性可以使用下面的公式：

(nodes_written + nodes_read) > replication_factor

例如，如果应用程序使用QUORUM级别来读和写，replication factor 值为3，那么，该设置能够确保2个节点一定会被写入和读取。读节点数加上写写点数（4）个节点比replication factor （3）大，这样就能确保一致性。

# 参考


1. http://www.datastax.com/documentation/cassandra/2.0/cassandra/dml/dml_config_consistency_c.html

# 并发控制

Cassandra并不支持类似关系型数据库的基于回滚和锁机制实现ACID的事务，Cassandra的类似事务的一致性，通过持久化事务，可调节的一致性和轻量级事务来实现。

# 轻量级事务

[1]尽管持久化事务和可调节的一致性已经可以满足许多用例，有一些情况，确实需要更强的原子控制。轻量级事务（lightweight transaction），又称（compare and set），使用线性化（linearizable）的一致性来满足需求。

例如，如果想确保一个用户帐号插入的唯一性，可以使用IF NOT EXISTS语句：

```
INSERT INTO customer_account (customerID, customer_email)
  VALUES ('LauraS', 'lauras@gmail.com')
  IF NOT EXISTS;
```

UPDATE也可以使用IF语句来比较一个或者多个字段的值：

```
UPDATE customer_account
  SET customer_email='laurass@gmail.com'
  IF customer_email='lauras@gmail.com';
```

在后台，Cassandra需要在发起事务的节点和集群中的其它节点之间有4次应答实现一个事务。所以，轻量级事务，对性能有影响。因此，只在绝对必要时，才应该使用轻量级事务，其他情况，一般都能通过可调节的一致性实现。

注意，SERIAL一致性级别 允许读取当前数据，包括uncommitted的数据。

# 原子化（**Atomicity**）

[2]在Cassandra里，写操作在行级别是原子化的，，也就是说，一次插入或者更新一行的多个列是一个原子操作。但是，因为Cassandra不支持类似关系型数据库的事务，Cassandra的写操作请求有可能返回失败，但实际上，数据已经写到某个replica了。

例如，如果写操作使用QUORUM级别，replication factor值为3， Cassandra需要写集群中所有节点，并且等待至少两个节点返回成功，如果某一个节点写失败，但是其他节点写成功，Cassandra会返回写失败，但是，那些写成功的节点的数据不会被回滚。

Cassandra使用timestamp来决定每一列的数据是否最新。timestamp是由客户端程序提供的。timestamp越近，代表值越新，如果有多个客户端同时更新一列数据，timestamp值最新的数据最终会被同步到所有节点。

# 持久性（**Durability**）

[3]Cassandra的写操作是durable的。对每一个节点的所有写操作，在返回成功之前，一定会确保保存到内存中的memtable和磁盘上的commit log。如果服务器在内存中的memtable数据还没保存到磁盘上之前宕机，等服务器重启时，commit log中的数据会被用来自动恢复重建memtable中的数据，所以，不会有数据丢失。

# 参考


1. http://www.datastax.com/documentation/cassandra/2.0/cassandra/dml/dml_ltwt_transaction_c.html
2. http://www.datastax.com/documentation/cassandra/2.0/cassandra/dml/dml_atomicity_c.html
3. http://www.datastax.com/documentation/cassandra/2.0/cassandra/dml/dml_durability_c.html

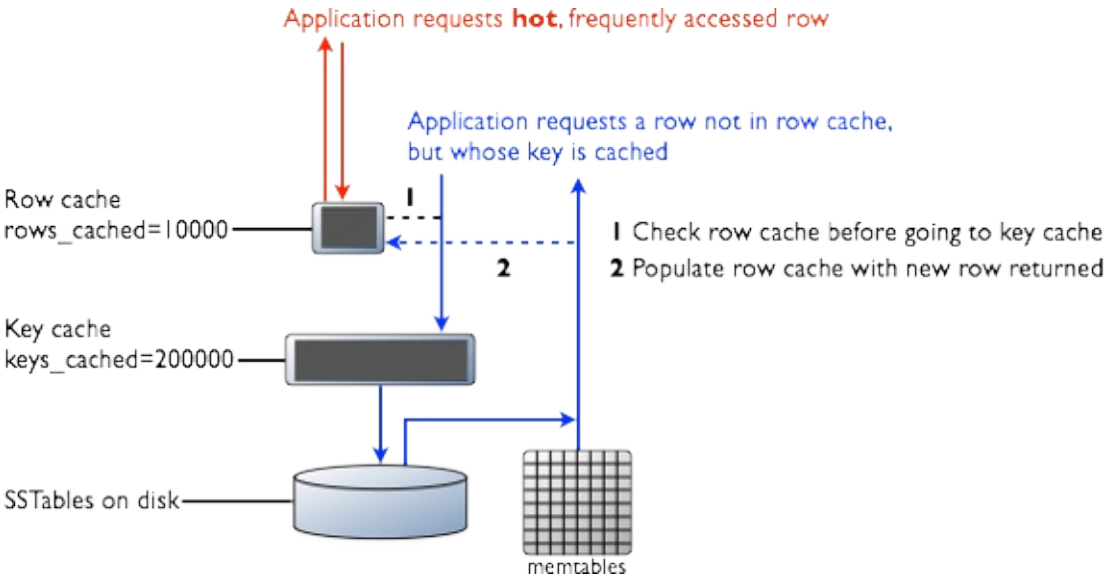
# 数据缓存

[1]Cassandra包含集成的分布式缓存支持。Cassandra的缓存内部会被定期保存到磁盘，机器重启时，缓存会从磁盘自动装载，因此，即使机器冷驱动，也不需要缓存预热。

**Cassandra**中有两个层次的缓存：

- Partition key缓存
- 行缓存

# 缓存是如何工作的？




[2]如上图，一个读请求因为有行缓存，无需磁盘搜索就返回了。另一个读操作，行缓存找不到数据，但是命中partition key缓存，在访问了SSTable之后，数据会被保存到行缓存，然后返回。

# 高效缓存使用心得

[3]下面是一些缓存使用的心得：

- 对于访问不频繁的数据，尽量少使用或不使用缓存；
- 部署尽量多的Cassandra节点，降低每个节点的负载；
- 对频繁读的数据，尽量在逻辑上分割成多个具体的表；

# 参考


1. http://www.datastax.com/documentation/cassandra/2.0/cassandra/operations/ops_configuring_caches_c.html
2. http://www.datastax.com/documentation/cassandra/2.0/cassandra/operations/ops_how_cache_works_c.html
3. http://www.datastax.com/documentation/cassandra/2.0/cassandra/operations/ops_cache_tips_c.html

# 客户端请求

[1]客户端的读写请求会平均分布到集群上的任意节点，因为Cassandra是一个点对点系统，每个节点都是平等的。

这一章，介绍客户端请求如何连接到一个节点，以及节点之间如何通信。

# 参考


1. http://www.datastax.com/documentation/cassandra/2.0/cassandra/architecture/architectureClientRequestsAbout_c.html

# 连接哪个节点？

一个客户端请求会连接集群上的哪个节点由客户端driver的配置决定。

例如，如果使用Datastax drivers，主要由下面这两个客户端配置决定连接到哪个节点：

- Contact points
- Load balancing policies

## Contact Points

Contact points设置的值是一个或者多个节点IP的列表。当driver在客户端创建Cluster实例时，driver首先会按照指定的顺序尝试连接contact points中的节点地址。如果第一个节点连接失败，则尝试连接第二个。只要任何一个节点连接成功，则停止尝试连接后面的节点地址。

为什么此时不需要连接所有的节点，是因为集群中的任何一个节点都保存了集群上所有节点的元数据，也就是说，只要连上一个节点，driver就能得到所有其他节点的配置信息。接下来，driver会使用获得到所有节点的配置信息构造连接池（构造连接池的时候，driver会连接所有的节点，但是，不是根据contact points中指定的节点地址，而是根据从服务端获得的所有节点的元数据）。这也意味着，contact points设置中，不必要指定集群中所有节点的地址。最佳实践是，在contact points中设置相对于客户端响应最快的所有节点的地址。

## Load Balancing Policies

默认情况下，一个客户端的Cluster实例管理了集群中所有节点的连接，并且，将客户端请求随机连接到任意节点。不过，在某些情况下，尤其是多数据中心时，默认行为的性能可能不是最优的。

例如，如果集群包含两个数据中心，一个在中国，一个在美国。如果客户端在中国，那么，应该避免直接连接美国的节点，不然太慢。

Cluster实例的Load balancing policies设置，决定了，客户端请求如何分配节点的连接。

Datastax driver中最常用的内置Load balancing policies是DCAwareLoadBalancePolicy。它可以指定客户端请求只连接到指定的数据中心的节点。

当然，你也可以实现自定义的load balanacing policies，实现自定义的客户端连接分配。

## Coodinator

[1]当客户端读写请求连接到某个节点时，这个被连接的节点被称作本次请求的coordinator。

一个coordinator的职责是客户端请求和真正拥有数据的节点之间的代理（Proxy）。coordinator根据服务端的分区器配置和replication策略设置来决定应该和集群中的哪些节点通信。

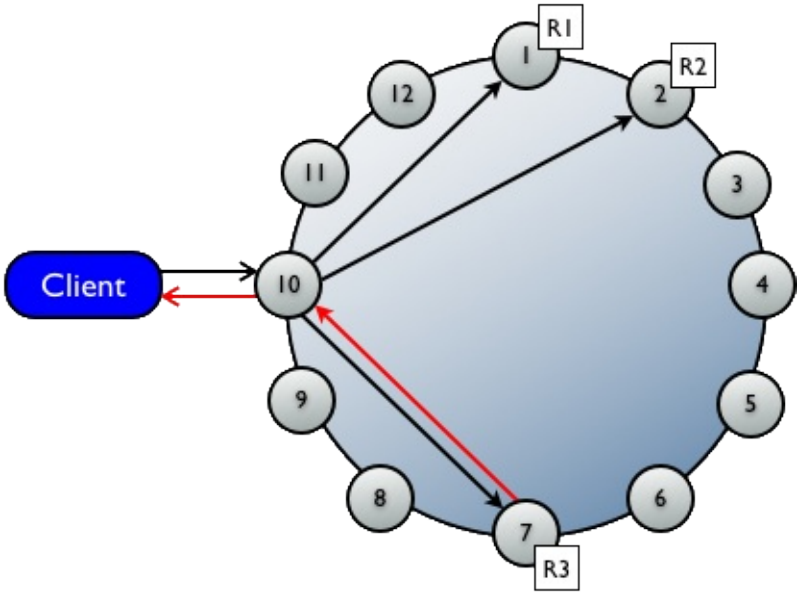
## 参考


1. http://www.datastax.com/documentation/cassandra/2.0/cassandra/architecture/architectureClientRequestsAbout_c.html

# 写请求

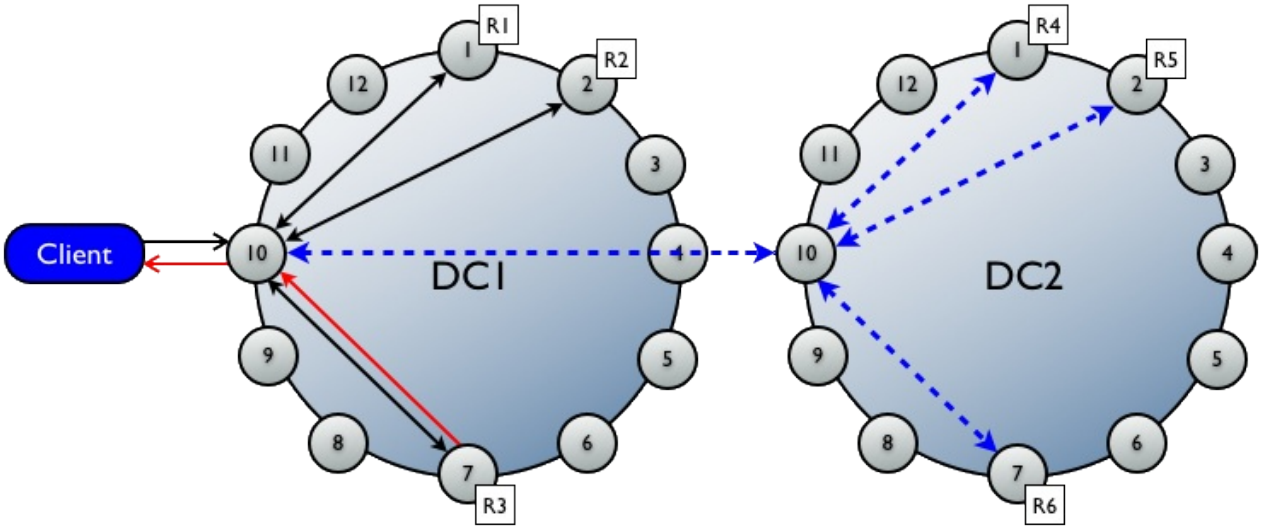
[1]如果所有的节点在同一个数据中心，coordinator会将写请求发到所有的replica。如果所有的replica节点可用，无论客户端指定的consistency level是什么，每一个replica都会处理写请求。写操作的consistency level只决定了coordinator在通知客户端写操作成功之前，多少个replica节点必须返回成功。

例如，一个单数据中心集群有10个节点，replication factor值为3，一个写请求，会被发给所有的三个replica节点。如果，客户端指定的consistency level是ONE，那么，只要任何一个replica节点通知coordinator写成功，coordinator就会返回客户端写成功。一个节点返回写成功，表示，写数据已经保存到这个节点的内存中的memtable和磁盘上的commit log。




[2]如果是多数据中心部署，Cassandra为了优化写性能，对于远程数据中心，会在每个远程数据中心的节点中，选择一个作为远程数据中心中的coordinator。因此，和远程数据中心里的replica nodes的通信，本地数据中心的coordinator，只需要和每个远程数据中心的这一个coordinator节点通信。

如果使用consistency level ONE或者LOCAL_QUORUM，coordinator只需要和本地数据中心的节点通信。因此，可以避免，跨数据中心的的通信影响写操作性能。




# 参考


1. http://www.datastax.com/documentation/cassandra/2.0/cassandra/architecture/architectureClientRequestsWrite.html
2. http://www.datastax.com/documentation/cassandra/2.0/cassandra/architecture/architectureClientRequestsMultiDCWrites_c.html

# 读请求

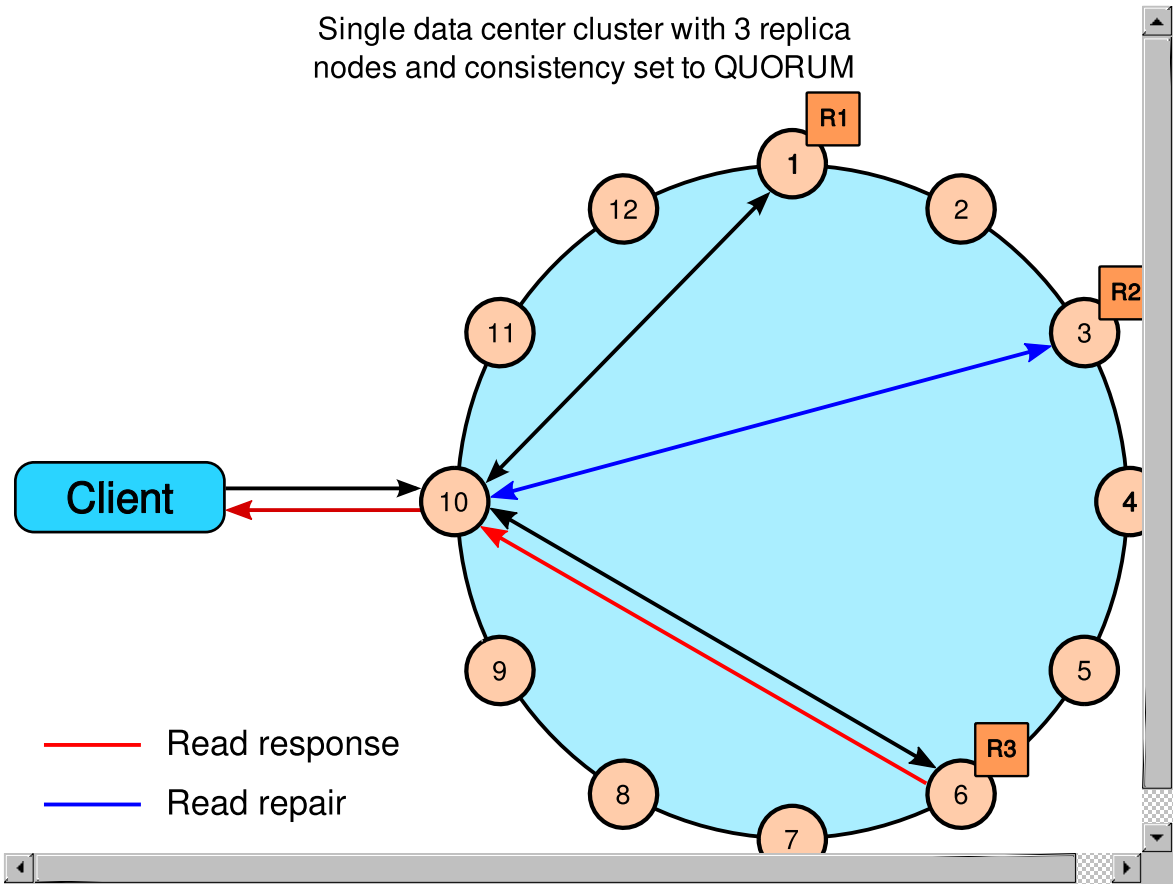
客户端请求需要读的replica数量，由客户端指定的consistency level决定。coordinator会将读请求发给响应最快的replica。如果多个节点返回了数据，coordinator会在内存中比较每一列的timestamp，返回合并后的最新的数据。

为了确保所有的replica对于经常访问的数据的一致性，在每一次读操作返回之后，coordinator会在后台同步所有其他replica上的该行数据，确保每个replica上拥有该行数据的最新版本。

# 举例

## 单数据中心，consistency level QUORUM

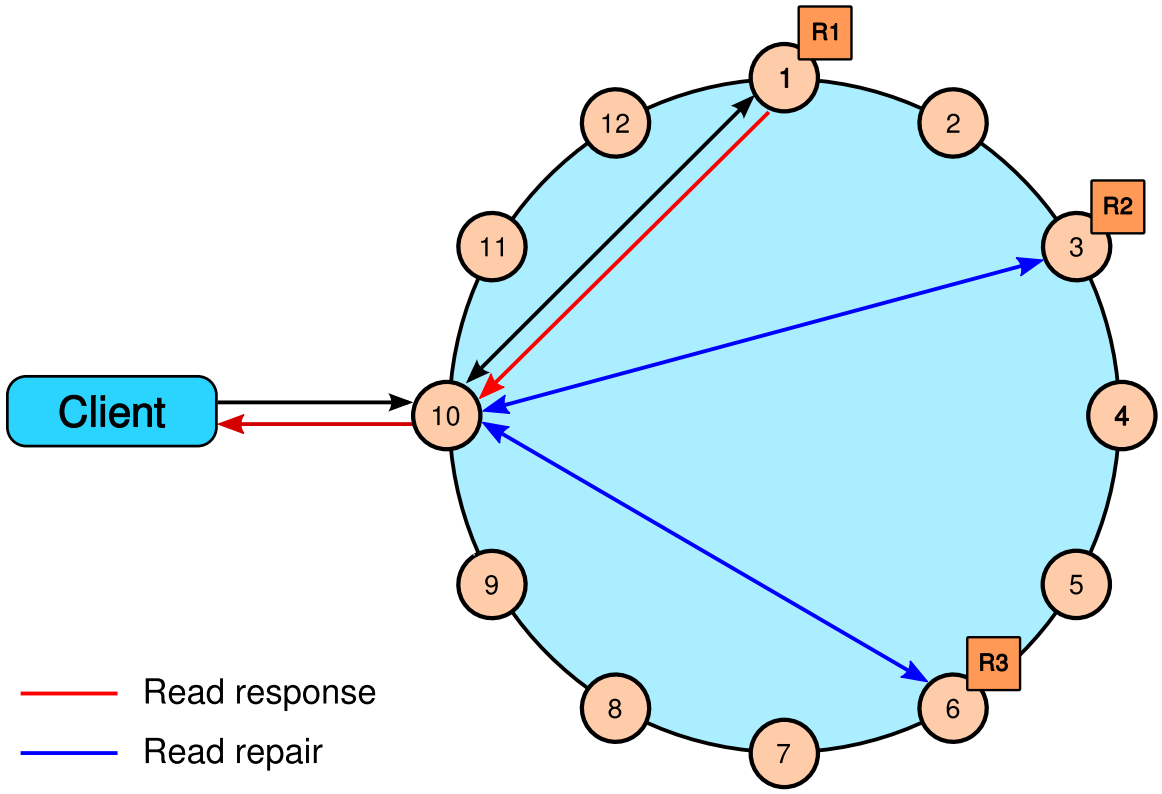
如果是单数据中心，replication factor值为3，读操作consistency level为QUORUM，那么，coordinator必须等待3个replica中的2个返回数据。 如果返回的数据版本不一致，合并后的最新的数据被返回。在后台，第三个replica的数据也会被检查，确保该行数据的最新版本在所有replica的一致性。




## 单数据中心，consistency level ONE

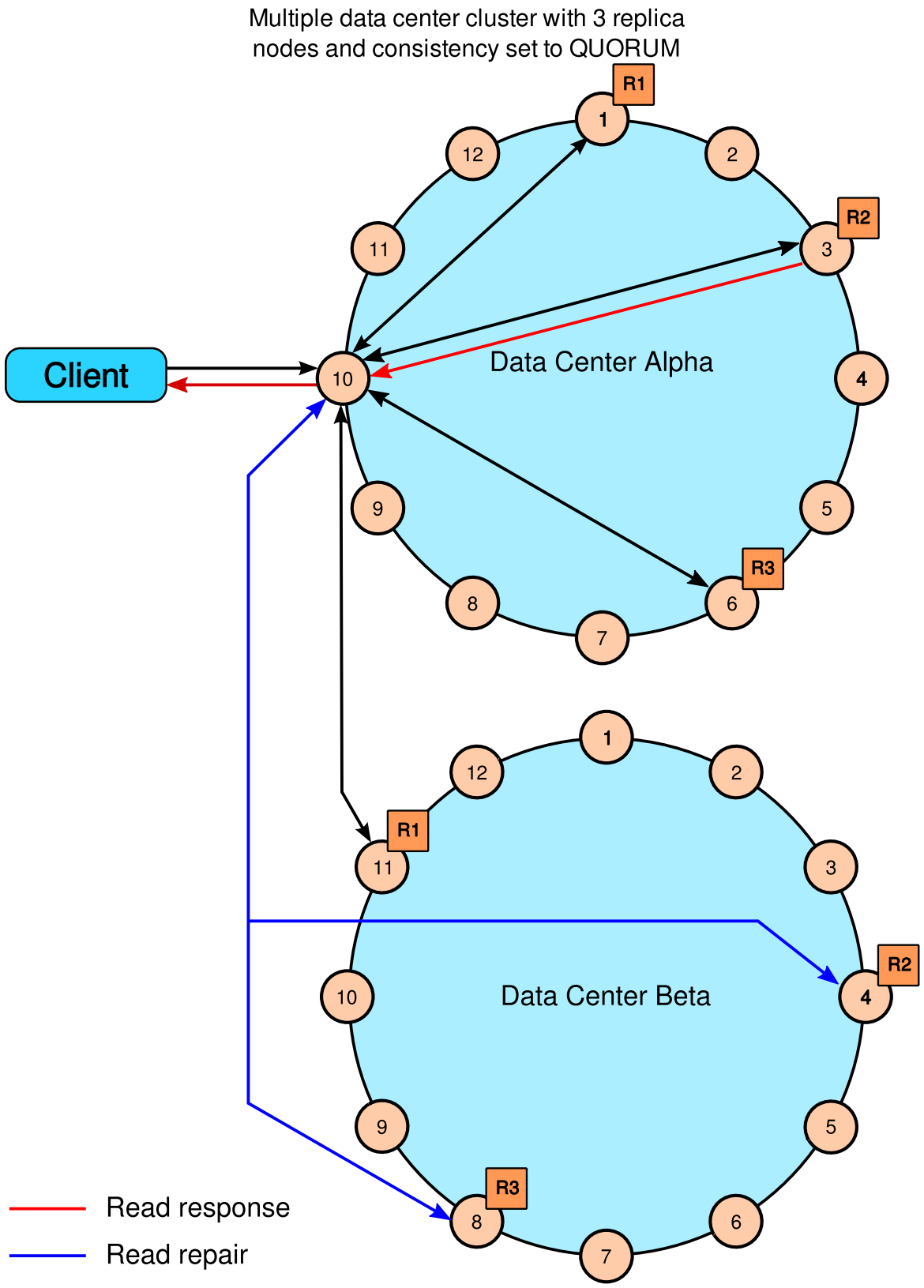
如果是单数据中心，replication factor值为3，读操作consistency level为ONE，coordinator访问并返回最近的replica返回的数据。在后台，其他两台replica的数据也会被检查，确保该行数据的最新版本在所有replica的一致性。

Single data center cluster with 3 replica nodes and consistency set to ONE



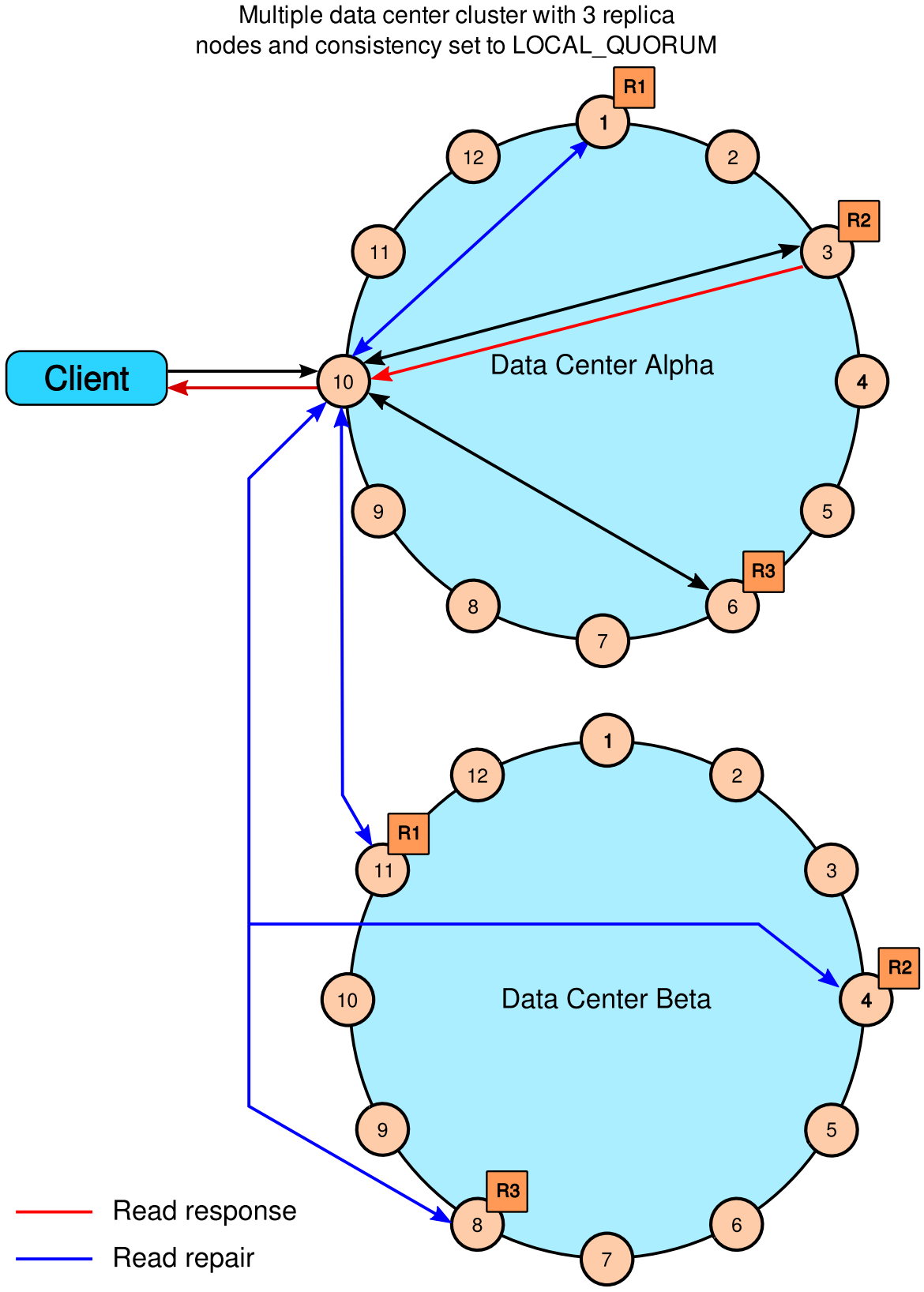

## 双数据中心，**consistency level QUORUM**

如果是双数据中心，replication factor值为3，读操作consistency level为QUORUM，coordinator必须等待4个replica返回数据。4个replica可以来自任意数据中心。在后台，其他所有数据中心的replica的数据也会被检查，确保该行数据的最新版本在所有replica的一致性。




## 双数据中心，**consistency level LOCAL_QUORUM**

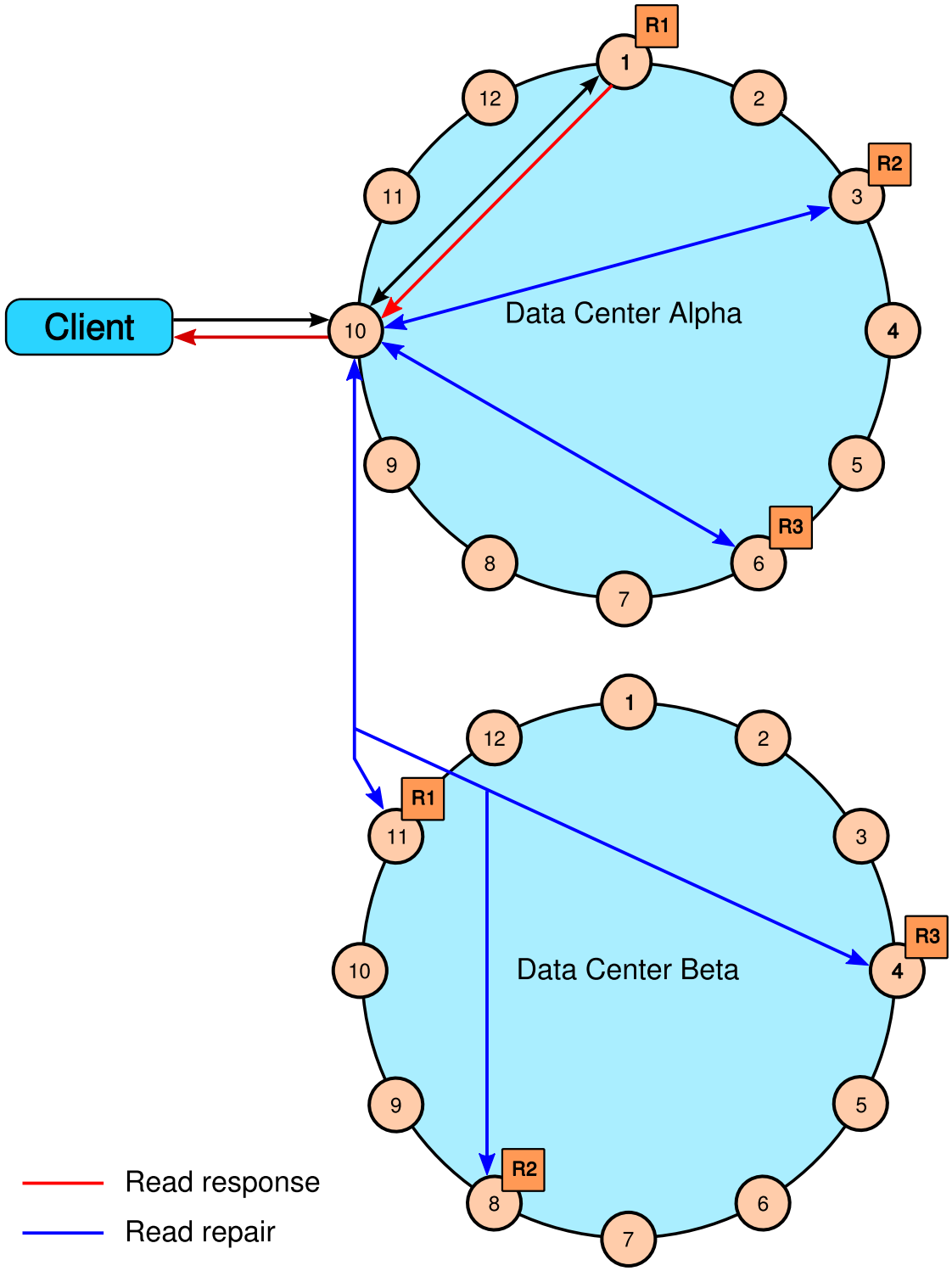
如果是双数据中心，replication factor值为3，读操作consistency level为LOCAL_QUORUM, coordinator必须等待本地数据中心的2个replica返回数据。在后台，其他所有数据中心的replica的数据也会被检查，确保该行数据的最新版本在所有replica的一致性。




## 双数据中心，**consistency level ONE**

如果是双数据中心，replication factor值为3，读操作consistency level为ONE, coordinator访问并返回最近的replica返回的数据，无论该replica是本地数据中心的还是远程数据中心的。在后台，其他所有数据中心的replica的数据也会被检查，确保该行数据的最新版本在所有replica的一致性。

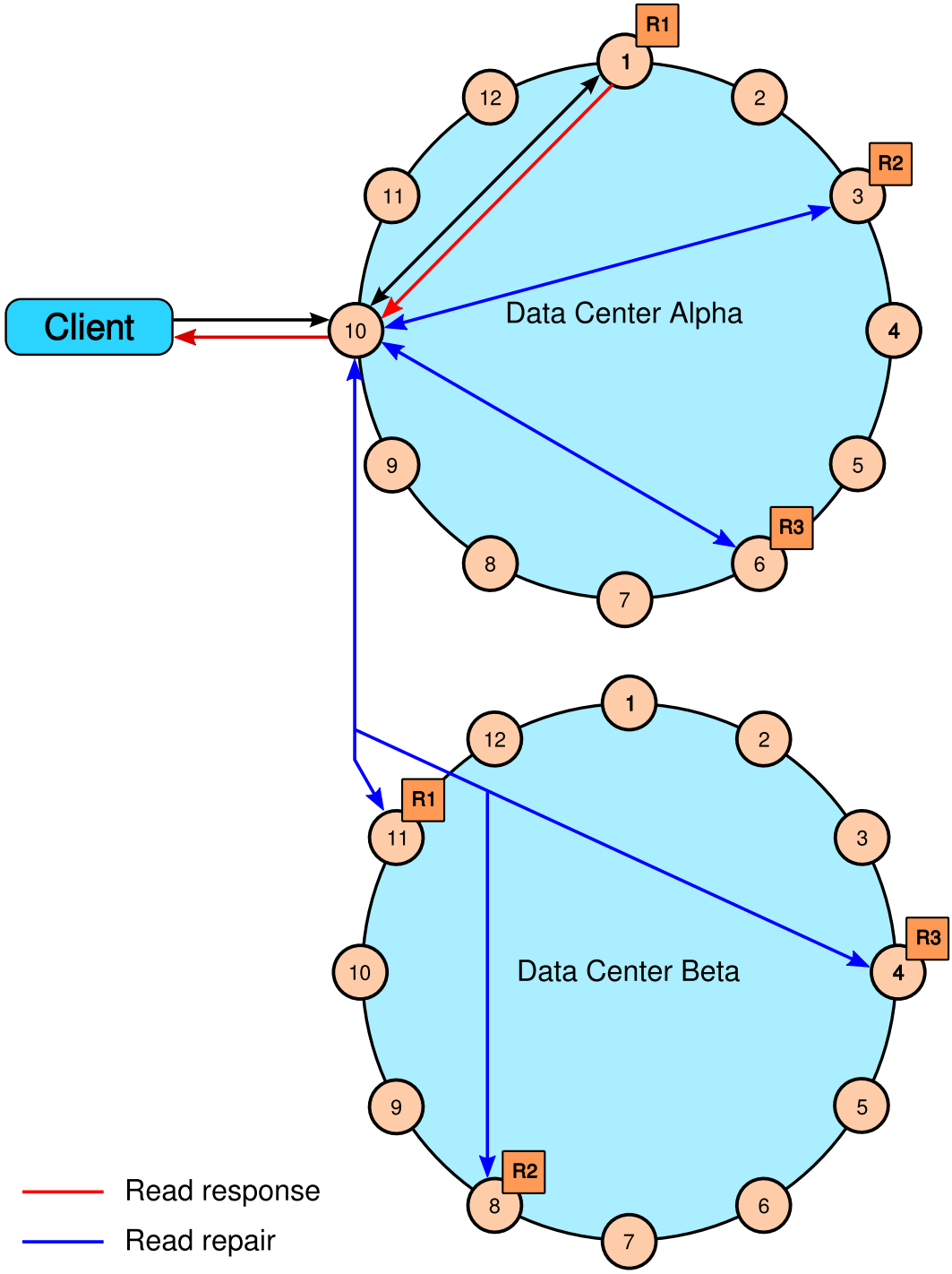
Multiple data center cluster with 3 replica nodes and consistency set to ONE




## 双数据中心，**consistency level LOCAL_ONE**

如果是双数据中心，replication factor值为3，读操作consistency level为LOCAL_ONE，coordinator访问并返回本地数据中心最近的replica返回的数据。在后台，其他所有数据中心的replica的数据也会被检查，确保该行数据的最新版本在所有replica的一致性。

Multiple data center cluster with 3 replica nodes and consistency set to LOCAL_ONE




## 使用speculative_retry做快速读保护（Rapid read protection）

快速读保护允许，即使coordinator最开始选择的replica节点down了或者超时了，依然能返回数据。如果一个表配置了speculative_retry参数，假如coordinator最先选择的replica读取超时，coordinator会尝试读取其他可用的replica代替。

Recovering from replica node failure with rapid read protection